Pascal Pons Follow

SOLVING CONNECT FOUR

Part 3 – MinMax algorithm

As a first step, we will start with the most basic algorithm to solve Connect 4.

The MinMax algorithm

Solving Connect 4 can been seen as finding the best path in a decision tree where each node is a Position. At each node player has to choose one move leading to one of the possible next positions. When it is your turn, you want to choose the best possible move that will maximize your score. But next turn your opponent will try himself to maximize his score, thus minimizing yours.

This leads to a reccursive algorithm to score a position. At each step:

- when it's your turn, the score is the maximum score of any of the next possible positions (you will play the move that maximizes your score),
- while when it's your opponent's turn, the score is the minimum score of next possible positions (your opponent will play the move that minimizes your score, and maximizes his).
- final positions (draw game after 42 moves or position with a winning alignment) get a score according to our score function defined in part 2

In practice exploring the full tree is most of the time untractable due to exponential growth of tree size with search depth. Most AI implementation explore the tree up to a given depth and use heuristic score functions that evaluate these non final positions. In this tutorial we will build a perfect solver and won't rely on heuristic scores. Thus we will explore the game until the end and our score function only gives exact score of final positions.

The Negamax variant

The Negamax variant of MinMax is a simplification of the implementation leveraging the fact that the score of a position from your opponent's point of view is the opposite of the score of the same position from your point of view.

Thus you can implement a single version of the recurssive function to compute a score of a position and no longer have to make the difference between you and your opponent.

Implementation

To implement the Negamax reccursive algorithm, we first need to define a class to store a connect four position. We will use a minimal interface allowing us to check if a column is playable, play a column, check if playing a column makes an alignment and get the number of moves played so far.

Here is a C++ definition of this interface, check the full source code for a basic implementation storing a position into an array.

```
/**
 * A class storing a Connect 4 position.
 * Function are relative to the current player to play.
 * Position containing aligment are not supported by this class.
 */
class Position {
  public:
    static const int WIDTH = 7;  // Width of the board
    static const int HEIGHT = 6; // Height of the board

    /**
     * Indicates whether a column is playable.
     * @param col: 0-based index of column to play
     * @return true if the column is playable, false if the column is already full.
     */
    bool canPlay(int col) const;

    /**
     * Plays a playable column.
     * This function should not be called on a non-playable column or a column making an alignment.
     *
     * @param col: 0-based index of a playable column.
     */
    void play(int col);

    /**
     * Indicates whether the current player wins by playing a given column.
     * This function should never be called on a non-playable column.
     * @param col: 0-based index of a playable column.
     * @return true if current player makes an alignment by playing the corresponding column col.
     */
    bool isWinningMove(int col) const;

    /**
     * @return number of moves played from the beginning of the game.
     */
    unsigned int nbMoves() const;
};
```

Then the Negamax function allowing to score any non final (without aligment) position is:

```
/*
 * Recursively solve a connect 4 position using negamax variant of min-max algorithm.
 * @return the score of a position:
 *  - 0 for a draw game
 *  - positive score if you can win whatever your opponent is playing. Your score is
 *    the number of moves before the end you can win (the faster you win, the higher your score)
 *  - negative score if your opponent can force you to lose. Your score is the oposite of
 *    the number of moves before the end you will lose (the faster you lose, the lower your score).
 */
int negamax(const Position &P) {
  if(P.nbMoves() == Position::WIDTH*Position::HEIGHT) // check for draw game
    return 0;

  for(int x = 0; x < Position::WIDTH; x++) // check if current player can win next move
    if(P.canPlay(x) && P.isWinningMove(x))
      return (Position::WIDTH*Position::HEIGHT+1 - P.nbMoves())/2;

  int bestScore = -Position::WIDTH*Position::HEIGHT; // init the best possible score with a lower bound of score.

  for(int x = 0; x < Position::WIDTH; x++) // compute the score of all possible next move and keep the best one
    if(P.canPlay(x)) {
      Position P2(P);
      P2.play(x);               // It's opponent turn in P2 position after current player plays x column.
      int score = -negamax(P2); // If current player plays col x, his score will be the opposite of opponent's score after playing col x
      if(score > bestScore) bestScore = score; // keep track of best possible score so far.
    }

  return bestScore;
}
```

This solver allows to compute the score of any non final position and not only its win/draw/loss outcome.

While it strongly solves Connect 4, the following benchmark shows that it is not at all efficient. We will see in the following parts of this tutorial how to optimize it step by step.

Benchmark

Here is the performance evaluation of this first basic implementation. You will note that this simple implementation was only able to process the easiest test set. It is able to process the same number of position per second than our reference benchmark, but it explores way to many positions.

Solver	Test Set name	mean time	mean nb pos	K pos/s
MinMax (strong solver)	End-Easy	790.28 µs	11,024	13,950
MinMax (strong solver)	Middle-Easy	N/A	N/A	N/A
MinMax (strong solver)	Middle-Medium	N/A	N/A	N/A
MinMax (strong solver)	Begin-Easy	N/A	N/A	N/A
MinMax (strong solver)	Begin-Medium	N/A	N/A	N/A
MinMax (strong solver)	Begin-Hard	N/A	N/A	N/A

mean time: average computation time (per test case). *mean nb pos*: average number of explored nodes (per test case). *N/A* means that the algorithm was too slow to evaluate the 1,000 test cases within 24h.

Tutorial plan

SOLVING CONNECT FOUR

Updated: February 19, 2017

Previous | **Next**